The Chemical Abstract Machine

Introduced in

> G. Berry and G. Boudol. *The chemical abstract machine.* Theoretical Computer Science, 96(1):217–248, 1992.

the *chemical abstract machine*, or *Cham*, is both a model of concurrency and a specific style of giving operational semantics definitions. Properly speaking, it is not an SOS definitional style. Berry and Boudol identify a number of limitations inherent in SOS, particularly its lack of true concurrency, and what might be called SOS's rigidity and *slavery to syntax*. They then present the Cham as an *alternative* to SOS.

In fact, like any of the other definitional styles seen in class so far, Cham is also going to be a particular definitional style *within* rewriting logic. That is, every Cham *is*, by definition, a specific

kind of rewrite theory; and Cham computation is precisely concurrent rewriting computation; that is, proof in rewriting logic.

The basic metaphor giving its name to the Cham is inspired by Banâtre and Le Mètayer's GAMMA language. It views a distributed state as a *solution* in which many *molecules* float, and understands concurrent transitions as *reactions* that can occur simultaneously in many points of the solution. It is possible to define a variety of chemical abstract machines. Each of them corresponds to a rewrite theory satisfying certain common conditions.

There is a common syntax, governed by some basic axioms, that is shared by all chemical abstract machines, with each machine possibly extending the basic syntax by additional function symbols and rules. The common syntax is typed, and can be expressed as

the following CFG Ω:

$$\begin{array}{rcl} \mathit{Molecule} & ::= & \mathit{Solution} \mid \mathit{Molecule} \triangleleft \mathit{Solution} \\ \mathit{Molecules} & ::= & \mathit{Set}_{,}[\mathit{Molecule}] \\ \mathit{Solution} & ::= & \{\!|\mathit{Molecules}|\!\} \end{array}$$

The operator $\_ \triangleleft \_$ is called the *airlock* operator, and $\{\!|\_|\!\}$ is called the *membrane* operator. This syntax is governed by some axioms. A first set of axioms state basically that *Molecules* are precisely sets of molecules, in the sense that the *concatenation of molecules*, "$\_, \_$", is an associative and commutative operation, with $\emptyset$ staying for the empty set of molecules. Another important axiom, the *airlock axiom*, is the following

$$\{\!|m, M|\!\} \rightleftharpoons \{\!|m \triangleright \{\!|M|\!\}|\!\}$$

where m ranges over *Molecule* and M over *Molecules*. The purpose of this axiom is to choose one of the molecules m in a solution as a

candidate for reaction with other molecules outside its membrane. Other axioms state that reactions can appear anywhere in a solution, *unconstrained by context*, and *concurrently*. We do not go into the formal details of what that means. For the time being, we use our intuition. Nevertheless, later on in the class we'll see that these axioms are both technically and intuitively captured by rewriting logic.

A *Cham* may add to the above chemical infrastructure additional syntactic categories, operators and "evolution" rules. Rules can be partitioned into the following intuitive categories, though no formal requirements impose a rule to be into one or another category:

- *Heating* rules, written using $\rightharpoonup$, are used to structurally rearange the solution so that reactions can take place.
- *Cooling* rules, written using $\rightharpoondown$, are used after reactions take place to remove useless molecules.

- *Reaction* rules, which change the solution in an irreversible way.

Rules may involve variables, but must obey some syntactic restrictions to avoid "multiset matching". The heating and Cooling rules can typically be paired, with each heating rule $t \rightharpoonup t'$ having a symmetric rule $t' \leftharpoonup t$, and vice-versa, so that we can view them as a single set of bidirectional *Heating/Cooling* rules, written $t' \rightleftharpoons t$. The airlock axiom is an example of such a heating/cooling bidirectional rule.

As one may expect, the reaction rules are the heart of the Cham and properly correspond to state transitions. The heating/cooling rules express *structural equivalence*, so that the reaction rules may apply after the appropriate structurally equivalent syntactic form is found. A certain strategy is typically given to address the problem of finding the right structural form, for example to perform "heating" as much as possible. In other words, we can view the reaction rules as being applied *modulo* the associativity and

commutativity of molecule composition and *modulo* heating/cooling rules (including the airlock).

As Berry and Boudol demonstrate in their paper, the Cham is particularly well-suited to give semantics to concurrent calculi and languages, yielding considerably simpler definitions than those afforded by SOS.

Since our example language is sequential, it cannot take full advantage of the Cham's true concurrent capabilities. Nevertheless, there are interesting Cham features that, as we explain below, turn out to be useful even in this sequential language application. A Cham semantics for our language is given below, following almost step-for-step the context reduction definition discussed before. One can formally show that a step performed using reduction under evaluation contexts is equivalent to a suite of heating steps followed by one reaction step and then by as many cooling steps as possible.

We distinguish two kinds of molecules:

- *syntactic molecules*, which are either evaluation contexts or redexes; and
- *store molecules*, which are pairs (x, i), where x is a variable and i is an integer.

The store is a solution containing store molecules. Definitions of evaluation contexts are translated into heating and cooling rules, bringing the redex to the top of the solution. This allows for the reduction rules to only operate at the top.

$$\{|s; a|\} \rightleftharpoons \{|s \rhd \{|\Box; a|\}|\}$$

$\{\!|x := a \rhd c|\!\} \rightleftharpoons \{\!|a \rhd \{\!|x := \Box \rhd c|\!\}|\!\}$

$\{\!|s_1; s_2 \rhd c|\!\} \rightleftharpoons \{\!|s_1 \rhd \{\!|\Box; s_2 \rhd c|\!\}|\!\}$

$\{\!|\{s\} \rhd c|\!\} \rightleftharpoons \{\!|s \rhd \{\!|\{\Box\} \rhd c|\!\}|\!\}$

$\{\!|\mathsf{if}\, b\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2 \rhd c|\!\} \rightleftharpoons \{\!|b \rhd \{\!|\mathsf{if}\, \Box\, \mathsf{then}\, s_1\, \mathsf{else}\, s_2 \rhd c|\!\}|\!\}$

$\{\!|a_1 + a_2 \rhd c|\!\} \rightleftharpoons \{\!|a_1 \rhd \{\!|\Box + a_2 \rhd c|\!\}|\!\}$

$\{\!|a_1 + a_2 \rhd c|\!\} \rightleftharpoons \{\!|a_2 \rhd \{\!|a_1 + \Box \rhd c|\!\}|\!\}$

(same for $*$, $-$, $/$, $\leq$, $\geq$, $=$, and, or)

$\{\!|\mathsf{not}\ b \rhd c|\!\} \rightleftharpoons \{\!|b \rhd \{\!|\mathsf{not}\ \Box \rhd c|\!\}|\!\}$

$\{\!|x \rhd c|\!\}, \{\!|(x, i) \rhd \sigma|\!\} \rightarrow \{\!|i \rhd c|\!\}, \{\!|(x, i) \rhd \sigma|\!\}$

$\{\!|i_1 + i_2 \rhd c|\!\} \rightarrow \{\!|i_1 +_{Int} i_2 \rhd c|\!\}$

(same for $*$, $-$, $/$, $\leq$, $\geq$, $=$, and, or)

$\{\!|\mathsf{not}\ t \rhd c|\!\} \rightarrow \{\!|t' \rhd c|\!\}$ where t and t' are opposite truth values

$$\{\!|x := i \rhd c|\!\}, \{\!|(x, i') \rhd \sigma|\!\} \to \{\!|\mathsf{skip} \rhd c|\!\}, \{\!|(x, i) \rhd \sigma|\!\}$$

$$\{\!|\mathsf{skip}; s_2 \rhd c|\!\} \to \{\!|s_2 \rhd c|\!\}$$

$$\{\!|\{\mathsf{skip}\} \rhd c|\!\} \to \{\!|\mathsf{skip} \rhd c|\!\}$$

$$\{\!|\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \rhd c|\!\} \to \{\!|s_1 \rhd c|\!\}$$

$$\{\!|\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \rhd c|\!\} \to \{\!|s_2 \rhd c|\!\}$$

$$\{\!|\mathsf{while}\, b\ s \rhd c|\!\} \to \{\!|\mathsf{if}\, b\, \mathsf{then}\, (s; \mathsf{while}\, b\ s)\, \mathsf{else}\, \mathsf{skip} \rhd c|\!\}$$

$$\langle s; a \rangle \to \{\!|s; a|\!\}, \{\!|\emptyset|\!\}$$

$$\{\!|\mathsf{skip}; a|\!\} \to \{\!|a \rhd \{\!|\Box|\!\}|\!\}$$

An Informal Guide to K

The purpose of this chapter is to explain K's basic intuitions to those who may be interested in using it but who may not like, may not have the background, or simply may not want to use the algebraic notation. Syntax is defined using only conventional and familiar context-free grammars, extended with lists and sets in a natural way. Operational intuitions are borrowed from rewriting logic, but no algebraic notation will be used.

Show Me an Example!

Here is an uncommented K definition of our simple language.

Computations and results:

$$AExp,\ BExp,\ Stmt,\ Pgm\ \leq K \tag{177}$$

$$Int,\ Bool\ \leq KResult \tag{178}$$

Configuration and intialization:

$$Config ::= Int \mid [\![Pgm]\!] \mid [\![Set[ConfigItem]]\!] \tag{179}$$

$$ConfigItem ::= k(K) \mid state(State) \tag{180}$$

$$[\![p]\!] = [\![k(p)\ state(\emptyset)]\!] \quad \text{for any } p \in Pgm; \emptyset \text{ is the empty state} \tag{181}$$

$$[\![k(i)\ state(\_)]\!] = i \quad \text{for any } i \in Int \tag{182}$$

Variable lookup:

$$k(\underbrace{\ x\ \rangle}_{\sigma[x]}\ state(\sigma) \quad \text{for any } x \in Var,\ \sigma \in State \tag{183}$$

Operators: (*ndstrict* stays for "non-deterministically strict")

$$\_ + \_\ [ndstrict\ extends\ \_ +_{Int} \_] \tag{184}$$

$$\_ - \_\ [ndstrict\ extends\ \_ -_{Int} \_] \tag{185}$$

$$\_ * \_\ [ndstrict\ extends\ \_ *_{Int} \_] \tag{186}$$

$$\_ / \_\ [ndstrict\ extends\ \_/_{Int} \_] \tag{187}$$

$$\_ \leq \_\ [ndstrict\ extends\ \_ \leq_{Int} \_] \tag{188}$$

$$\_ \geq \_\ [ndstrict\ extends\ \_ \geq_{Int} \_] \tag{189}$$

$$\_ = \_\ [ndstrict\ extends\ \_ =_{Int} \_] \tag{190}$$

$$\_ \mathsf{and} \_\ [ndstrict\ extends\ \_ \mathsf{and}_{Bool} \_] \tag{191}$$

$$\_ \mathsf{or} \_\ [ndstrict\ extends\ \_ \mathsf{or}_{Bool} \_] \tag{192}$$

$$\mathsf{not} \_\ [strict\ extends\ \mathsf{not}_{Bool} \_] \tag{193}$$

Statements:

$$\mathsf{skip}\ [.] \tag{194}$$

$$\_ := \_\ [strict(2)] \tag{195}$$

$$k(\underline{x := i\rangle}_{\cdot}\ state(\underline{\quad\sigma\quad}_{\sigma[x \leftarrow i]}) \quad \text{for any } x \in Var,\ i \in Int,\ \sigma \in State \tag{196}$$

$$\_\,;\_\ [strict(1)\_] \text{ (captures ";" for both } Stmt \text{ and } Pgm) \tag{197}$$

$$\{\_\}\ [strict\ .] \tag{198}$$

$$\mathsf{if}\ \_\ \mathsf{then}\ \_\ \mathsf{else}\ \_\ [strict(1)] \tag{199}$$

$$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_1 \tag{200}$$

$$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_2 \tag{201}$$

$$k(\underline{\qquad\qquad \mathsf{while}\ b\ s \qquad\qquad\rangle}_{\mathsf{if}\ b\ \mathsf{then}\ s; \mathsf{while}\ b\ s\ \mathsf{else}\ .}) \tag{202}$$

OK, What is K?

K is a rewriting-based framework for computations. K was introduced in an optimized but algebraic form in the context of the Maude executable rewriting logic language for the first time in

> G. Rosu. *Programming Language Design* course (CS322) at the University of Illinois at Urbana-Champaign, 2003.

By saying that K is a "rewriting-based" framework, in contrast to a "reduction-based" one, we mean that in K rules can be applied concurrently and unrestricted by context, following the basic intuitions and operational/semantical strengths of rewriting logic.

What is the Philosophy of K?

When defining a language in K, one needs to define three things:

- A *configuration structure*, serving as a backbone for storing everything we need, such as computations, threads, states, environments, etc. It can have various layers of information and configuration items are allowed to multiply themselves. We had a flat, two-item configuration in our example above.
- *Computation structural equations*, allowing one to change representations of computations at will. Each fragment of program will have a unique computation equivalence class, which can be derived using the computation equations.
- *Rewriting rules* and *equations*. Rules capture the execution steps of the defined language. Equations capture structural rearangements that do not count as execution steps.

Computations extend part of the syntax of language; one must explicitly say which syntactic categories are intended to be computations (177), and which are intended to be results of computations (178). The syntactic categories which were extended into computations, as well as the original language constructs, do not exist anymore from here on. They are completely replaced by computations and computation constructs; however, we still use generic names, such as a, b, s, or p, to refer to computations originating from *AExp*, *BExp*, *Stmt*, or *Pgm*, respectively.

To start the machinery, one needs to create an initial configuration with the given program (181). Typically, the program is placed somewhere in the configuration where a computation is expected. If everything is ok (program terminates, no "runtime errors", etc.), the program will eventually become a result. When that happens, one can disolve the configuration and keep the result (182).

For this trivial language, all the computation structural equations

were defined implicitly by the strictness attributes, so the user sees none. Attribute *strict* says that the operator is strict in its arguments sequentially from left to right, and *ndstrict* says that it is non-deterministically strict in its arguments, in the sense that its arguments can be evaluated in any order. A number or a list of numbers passed to a strictness attribute says that it applies only to those arguments. For example, the conditional is strict only in its first argument. Computation structural equations know how to "pass in front" of the computation fragments of program that need to be evaluated, and also how to "plug them back" once evaluated. Specifically, the *ndstrict* attribute of $\_ + \_$ is equivalent to the following two equations in K (a_1 and a_2 are computations in K):

$$a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2$$

$$a_1 + a_2 \rightleftharpoons a_2 \curvearrowright a_1 + \square$$

To distinguish them from the other equations, we use the symbol

"$\rightleftharpoons$" instead of "$=$" in computational structural equations. Computations have a monoid structure, with unit a central dot "$\cdot$" and with composition $\_ \curvearrowright \_$, called "computation sequencing".

Rewrite rules and equations can be given either explicitly, such as (181), (182), (200), and (201), or implicitly through specialized K-notation. A beginer would probably prefer explicit equations and rules, but an expert can take advantage of K's compact notation and move quickly to the interesting aspects of a language definition.

Here is the intuition behind the major K notations:

- ***K Attributes.*** The arguments of a language construct can be put in two categories: those that the operator is strict in and those that the operator is not strict in. Attributes *strict* and *ndstrict*, or their absence, give us this information. The *extends* attribute corresponds to a *rewrite rule* saying what to do after the strict arguments were evaluated. A construct may have

more than one *extends* attribute; a rewrite rule will be generated for each. The *extends* attribute for addition, for example, corresponds to the rule

$$i_1 + i_2 \rightarrow i_1 +_{Int} i_2 \text{ where } i_1, i_2 \in Int.$$

The default attribute of a construct, if any, tells, via an implicit *equation*, what to do with the non-strict arguments. The default attribute is not preceeded by any special keyword. In this example we have three such default attributes, a "." in (194) and (198), and a "_" in (197). The meaning of the "." is "nothing", or "forget it", or "disolve it". Specifically, (194) and (198) are equivalent to the following two equations:

$$\mathsf{skip} = .$$

$$\{s\} = s \quad \text{where } s \text{ is a computation}$$

If one wanted to first evaluate the body of the block and then

get rid of the block in one execution step, as we did when we defined the same language in our previous definitional styles, then one would replace (198) by "{_} [*strict extends* .]" or even by "{_} [*strict extends* skip]". The meaning of "_" is "whatever is there". In general, these atributes can be any computation fragments; underscores stay for the expected arguments in the expected order (one can also use argument names in case underscores are ambiguous). For example, (197) is equivalent to saying "evaluate first argument and keep the second *as is* in the computation", which translates into the equation:

$$s; k = s \curvearrowright k \quad \text{where } s \text{ and } k \text{ are computations}$$

An even more compact way of writing (198) would have been "{_} [_]", saying that the operation is not strict, so nothing is scheduled for evaluation, and the remaining argument (its only one) stays *as is*.

Obviously, the number of arguments in both *extends* and default attributes must match the strictness information.

- ***K Contexts.*** To avoid repeating context information that does not change in the left-hand and and the right-hand -sides of equations and rules, we take the liberty to use the two-dimensional notation

$$\begin{array}{c} C[\underline{t_1}, \underline{t_2}, ..., \underline{t_n}] \\ t'_1 \;\; t'_2 \qquad t'_n \end{array}$$

for *rules* $C[t_1, t_2, ..., t_n] \rightarrow C[t'_1, t'_2, ..., t'_n]$, and similarly

$$\begin{array}{c} C[\underline{t_1}, \underline{t_2}, ..., \underline{t_n}] \\ t'_1 \;\; t'_2 \qquad t'_n \end{array} \quad \text{(dotted horizontal lines)}$$

for *equations* $C[t_1, t_2, ..., t_n] = C[t'_1, t'_2, ..., t'_n]$. When using K in ASCII, we typically replace the horizontal lines by $\Rightarrow$, so we

write $C[t_1 \Rightarrow t'_1,\ t_2 \Rightarrow t'_2,\ ...,\ t_n \Rightarrow t'_n]$; to distinguish between equations and rules we prefix this notation with a keyword.

- ***K Underscore Variables and Angle Brackets.*** The K-context notation above allows two other notation optimizations. Like in Prolog, we use an underscore "_" as a variable matching anything that we don't care (if we used rules then we would care of anything, because we would have to repeat it in the right-hand-side. With the K-context and the underscore variable notations, the rule for assignment in our language would be

$$k(\underline{x := i} \curvearrowright \_)\ state(\underline{\quad \sigma \quad}) \qquad \text{for any integer number } i$$
$$\cdot \qquad\qquad\qquad\qquad \sigma[x \leftarrow i]$$

The more compact notation in (196) uses the K angle bracket convention which can be used whenever the enclosed structure is a list or a set. The angle bracket reads "and so on".

Like most notations, the K-notation may look strange, obscure and even artificial at first sight. One is, of course, free to use it or not to use it. K's notational conventions were not designed in an *adhoc* manner, but through a series of iterations and refinements, defining large languages in K and observing that many feature definitions followed similar patterns, or that large portions of a rule were redundant. Repetitive definitions and redundance are not only boring and heavy looking, but, more importantly, they are error prone and non-modular. Nevertheless, if one does not want to use our optimized K-notation then one can simply ignore it. The philosophy of K stays not in its specialized notation, but in its approach to define programming languages by "swallowing" program syntax into computation structures, which can then be reduced by means of ordinary, context-insensitive term rewriting. Below we give the K definition of the language above (skiping the preamble defining the configuration which remains unchanged)

using none of K's specialized notations. ($x \in \mathit{Var}$, $\mathit{rest}, a_1, a_2, s_1, s_2, a, b, s, k \in K$, $i_1, i_2 \in \mathit{Int}$, $t \in \{\mathsf{true}, \mathsf{false}\}$)

$$k(x \curvearrowright \mathit{rest})\ \mathit{state}(\sigma) \to k(\sigma[x] \curvearrowright \mathit{rest})\ \mathit{state}(\sigma) \tag{203}$$

$$a_1 + a_2 \rightleftharpoons a_1 \curvearrowright \square + a_2 \tag{204}$$

$$a_1 + a_2 \rightleftharpoons a_2 \curvearrowright a_1 + \square \tag{205}$$

$$i_1 + i_2 \to i_1 +_{\mathit{Int}} i_2 \tag{206}$$

(similarly for the other binary operators)

$$\mathsf{not}\ \ b \rightleftharpoons b \curvearrowright \mathsf{not}\ \square \tag{207}$$

$$\mathsf{not}\ \ t \to \mathsf{not}_{\mathit{Bool}}\ t \tag{208}$$

$$\mathsf{skip} = . \tag{209}$$

$$x := a \rightleftharpoons a \curvearrowright x := \square \tag{210}$$

$$k(x := i \curvearrowright \mathit{rest})\ \mathit{state}(\sigma) \to k(\mathit{rest})\ \mathit{state}(\sigma[x \leftarrow i]) \tag{211}$$

$$s; k \rightleftharpoons s \curvearrowright k \tag{212}$$

$$\{s\} \rightleftharpoons s \tag{213}$$

$$\mathsf{if}\ b\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \rightleftharpoons b \curvearrowright \mathsf{if}\ \square\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2 \tag{214}$$

$$\mathsf{if\ true\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_1 \tag{215}$$

$$\mathsf{if\ false\ then}\ s_1\ \mathsf{else}\ s_2 \rightarrow s_2 \tag{216}$$

$$k(\mathsf{while}\ b\ s \curvearrowright rest) = k(\mathsf{if}\ b\ \mathsf{then}\ s; \mathsf{while}\ b\ s\ \mathsf{else}\ .\ \curvearrowright rest) \tag{217}$$

What is Rewriting Logic?

We are going to discuss *rewriting logic (RL)* in more depth later in this course. Here we only informally discuss rewriting logic's very basic features that are needed to understand how K works. Simply put, RL is a very intuitive and simple computational logical framework. A *RL theory* consists of:

- *Syntax*, defining syntactic categories by means of "sorts" and operators on them, which allows us to define "uninterpreted" *terms*; in this section we use CFGs to define syntax;
- *Equations*, defining structural identities on terms, in the sense that equal terms can be substituted for each other unrestricted in any context. Intuitively, equal terms are *the same thing represenced differently*, the same way π, 3.1459..., or the circumference of a circle of diameter 1 are the same thing; and

- *Rewrite rules*, defining *irreversible transitions between terms*.

Both equations and rules can be parametric in rewriting logic, in the sense that they can contain variables that act as placeholders for subterms that *match* them. To avoid defining substitutions and matching in equational and rewriting logic, in this section, like in our language definitions using the other definitional styles, by a "parametric" equation or rule we mean a recursively enumerable set of equations or rules, one per each parameter instance that satisfies the side conditions.

Unlike in reduction semantics with evaluation contexts, there are no context restrictions on applications of equations and/or rules (extensions of RL have been proposed recently allowing some forms of context-sensitivity, but we are not going to use those). This allows for RL to serve as a very simple and elegant foundation for *true concurrency*, because rules and equations can indeed be

applied concurrently.

And What? Show Me K!

In K, we define programming languages as rewrite logic theories whose syntax captures the syntax of the defined programming language together with any other auxiliary operators needed, whose equations capture structural rearrangements of computation structures that do not count as computations in the defined language, and whose rules capture the intended computational steps of the defined language.

In K, *computations are first-class-citizen structures*, or better say *terms*, in the sense that they can be handled like any other terms in a rewrite theory: they can be rewritten into other computations, can be passed as arguments of operations and thus rewritten in context, and so on. Intuitively, computations are structures derived

from the original program or fragments of it that contain computational meaning, that is, that can be potentially evaluated if allowed or asked to do so in a particular context. In particular, the original syntax of the programming language provides a set of obvious candidates for computation constructors. If one wants to regard certain syntactic categories as computations (this will always happen in K language definitions), then one should do it explicitly as follows:

$$AExp,\ BExp,\ Stmt,\ Pgm \leq K$$

We prefer to read the above notation as follows: "syntactic categories *AExp*, *BExp*, *Stmt* and *Pgm* sink into *K*". From this high-level user-friendly annotation, one can derive the explicit syntax for computations as follows.

First, one adds automatically to the grammar of computations all the language constructs for *AExp*'s, *BExp*'s, *Stmt*'s, and *Pgm*'s,

replacing *AExp*, *BExp*, *Stmt* and *Pgm* with K everywhere. In other words, the syntax of our language is being "swallowed" by computations. One can also define different computation types for different syntactic categories, for example $AExp \leq K_{AExp}$, $BExp \leq K_{BExp}$, etc., but then the syntax for computations becomes slightly more involved.

K provides two builtin types of constructors for computations, which are therefore always assumed by default as part of the syntax for computations, namely

- $\_ \curvearrowright \_$ for *sequencing of computations*; and
- $f(K, K, \ldots, K)$ for *freezing of computations*, where f is any identifier, serving as the name of the freezer, and its arbitrarily many arguments are the frozen computations.

The following CFG shows the complete computation syntax that can be automatically derived from our user-friendly notation above:

$$
\begin{array}{rcl}
K & ::= & List[K]_{\curvearrowright} \mid Id\,(List[K]) \\
& \mid & K + K \mid K - K \mid K * K \mid K/K \\
& \mid & K \leq K \mid K \geq K \mid K \;=\; K \mid \\
& \mid & K \text{ and } K \mid K \text{ or } K \mid \text{not } K \\
& \mid & Var := K \mid K; K \mid \text{if } K \text{ then } K \text{ else } K \mid \text{while } K\,K
\end{array}
$$

$List[K]_{\curvearrowright}$ stays for "lists of K structures with $\curvearrowright$ as list concatenation". Lists and/or (multi-)sets of elements of any structures are assumed defined whenever needed in K. We regard lists as associative monoids and sets as both associative and commutative monoids in K. The default empty list or set is a central dot "$\cdot$", the default list concatenation is a comma "," and the default set concatenation is a space (written "_ _"). One can change the unit using a superscript and the concatenation symbol using a subscript. For example, $List[Int]_{:}^{nil}$ stays for lists of integers separated by ":", with nil the empty list. Lists and sets can be

easily defined using CFGs, or algebraic specifications, or many other formalisms, so we do not bother defining them formally here.

To reduce the number of parentheses in computations, we assume that every other operator binds tighter than $\_ \curvearrowright \_$.

As discussed later in the class, there is an automatic way to *faithfully* capture context reduction definitions in K. By "faithfully" we mean that everything that can be done with the context reduction definition, can be done, in an isomorphic correspondence, in the resulting K definition. This way, context reduction can be regarded as a definitional methodology within K. Nevertheless, it will often be more convenient to define languages directly in K, using K's full strength.

I Said, "Show Me K"!

After sinking the language syntax into computations, the very first thing one needs to do is to define the *computation (structural) equations*, which allow us to regard computations many different but completely equivalent ways. For example, a computation $a_1 + a_2$ may be regarded also as $a_1 \curvearrowright f_1(a_2)$, with the intuition "schedule a_1 for evaluation and freeze a_2 in freezer f_1", but also as $a_2 \curvearrowright f_2(a_1)$ (if addition is intended to be non-deterministic, like in our language). The role of the freezers is to store the remaining computations for future processing.

Recall that we can freeze computations at will in K, using freezers like f_1 and f_2 above. In complex K definitions, we may need many computation freezers, making definitions look heavy and hard to read. Therefore, we adopt the following *freezer naming convention*:

If a computation can be seen as $c[k, k_1, ..., k_n]$ for some multicontext c and a freezer is introduced to schedule k for computation, then the name of the freezer is $c[\Box, \_, ..., \_]$.

Additionally, to increase readability, we take the freedom to "plug" the remaining computations in the freezer name, that is, we write $c[\Box, k_1, ..., k_n]$ instead of $c[\Box, \_, ..., \_](k_1, ..., k_n)$ With this convention, the addition computation $a_1 + a_2$ can be regarded both as $a_1 \curvearrowright \Box + a_2$ and as $a_2 \curvearrowright a_1 + \Box$. Note that "$\Box$" is not a "hole", but just part of the name of the freezer. Nevertheless, it can almost be thought of as a "hole" (if the sequentialization of computations, $\curvearrowright$, is thought of as a "plug").

In K definitions, one typically defines zero, one, or more computation equations per language construct, depending on its intended evaluation strategy. The idea is to "schedule for evaluation" any of its allowed subcomputations. With this in mind, we can write the following structural equations for our language,

where $K^\circ \subseteq K$ *is the set of computations containing no freezer*:

Computation structural equations:

$a_1 + a_2 = a_1 \curvearrowright \square + a_2$ where $a_1 \in K^\circ$ and $a_2 \in K$

$a_1 + a_2 = a_2 \curvearrowright a_1 + \square$ where $a_1 \in K$ and $a_2 \in K^\circ$

(... similarly for $-$, $*$, $/$, $\leq$, $\geq$, and , or)

not $b = b \curvearrowright$ not $\square$ where $b \in K^\circ$

$x := a = a \curvearrowright x := \square$ where $x \in Var$ and $a \in K^\circ$

$s_1; s_2 = s_1 \curvearrowright \square; s_2$ where $s_1 \in K^\circ$ and $s_2 \in K$

$\{s\} = s \curvearrowright \{\square\}$ where $s \in K^\circ$

if b then s_1 else $s_2 = b \curvearrowright$ if $\square$ then s_1 else s_2 where $b \in K^\circ$ and $s_1, s_2 \in K$

Note that each time a computation was "scheduled" for evaluation it was required to contain no freezers, that is, to be in K°. That requirement enforces a cartain discipline in how these equations can

be applied. Indeed, they can be iteratively applied from left to right scheduling for evaluation any allowed subexpressions, and then they can be applied from right to left plugging back the subexpressions in reversed order. This discipline guarantees the following crucial property of computations:

> ***Proposition:*** *Each computation equivalence class contains at most one program or fragment of program.*

For each program or fragment of program e, we let $\hat{e}$ denote the *computation (equivalence) class of* e, that is, the set of all computations that can be derived from e using the computation equations and equational logic. For example, here is the

computation equivalence class of $x * (y + 2)$:

$$
\begin{aligned}
& x * (y + 2) \\
& x \curvearrowright \square * (y + 2) \\
& x \curvearrowright \square * (y \curvearrowright \square + 2) \\
& x \curvearrowright \square * (2 \curvearrowright y + \square) \\
& y + 2 \curvearrowright x * \square \\
& y \curvearrowright \square + 2 \curvearrowright x * \square \\
& 2 \curvearrowright y + \square \curvearrowright x * \square \\
& x * (y \curvearrowright \square + 2) \\
& x * (2 \curvearrowright y + \square)
\end{aligned}
$$

Note that the computation equations only had to apply from left-to-right starting with the original expression in order to derive its entire computation class. Because of the K° restriction, these

equations can apply from right-to-left only on positions that were previously expanded using the same equation from left-to-right. The following computations could have also been derivable equationally if we had dropped the restriction that the scheduled computation must be in K°:

$$y \curvearrowright x * (\Box + 2)$$

$$2 \curvearrowright x * (y + \Box)$$

$$y \curvearrowright x \curvearrowright \Box * (\Box + 2)$$

$$2 \curvearrowright x \curvearrowright \Box * (y + \Box)$$

$$(y \curvearrowright x) * (\Box + 2)$$

$$(2 \curvearrowright x) * (y + \Box)$$

$$\Box + 2 \curvearrowright (y \curvearrowright x) * \Box$$

$$y + \Box \curvearrowright (2 \curvearrowright x) * \Box$$

While one could say that these alternative computations still make

sense, especially the first two above, we prefer to call them *junk computations*, because they can lead to very undesired results. Supposing that the K° requirement is dropped, one can show that two different expressions have the same computation class:

Exercise 16 *Suppose that for a language construct* _@_ *whose arguments can be evaluated non-deterministically, one defines the following computation structural equations:*

$$k_1 @ k_2 = k_1 \curvearrowright \Box @ k_2 \quad \textit{where } k_1, k_2 \in K, \textit{and}$$
$$k_1 @ k_2 = k_2 \curvearrowright k_1 @ \Box \quad \textit{where } k_1, k_2 \in K.$$

Then show that $\widehat{(x@y)@(u@v)} = \widehat{(y@x)@(v@u)}$.

(Hint: it may be easier to implement a program that shows that by exhaustive search of computations; using Maude, one can define two rules per equation, one from left to right and the other from right to left, and then use a `search` *command to search for a path*

going from one computation to the other).

The exercise above shows that the restriction to require the scheduled computation to contain no freezing operators is necessary. Indeed, if we take @ to be division, then the two computations would produce entirely different results even when x, y, u and v are constants!

For this language, the computation equations above capture the same intuition as that captured with the definition of evaluation contexts in context reduction: they give for each language construct the subexpressions which are allowed to be evaluated. As already mentioned, we'll discuss a technique to automatically transform a context reduction definition into a K definition; the K equations above could be derived from the grammar of evaluation contexts, using a general procedure that will be discussed in a later class.

There is also an interesting analogy with CHAM, the "chemical

abstract machine" of Berry and Boudol:

> G. Berry and G. Boudol. *The chemical abstract machine.* Theoretical Computer Science, 96(1):217–248, 1992,

If one regards computations as "chemical solutions", then what we call "computation structural equation" above can be thought of as a pair of rules "heating/cooling" in CHAM. Recall that the role of "heating" in CHAM is to rearrange the solution so that "reactions" can take place. Once reactions take place, "cooling" rules insert the result of the reaction back into the solution. Therefore, reactions in CHAM take place "modulo" heating and cooling. Of course, multiple reactions can take place at the same time, which is what makes CHAM an elegant model for true concurrency. Similarly, rewrites in K definitions take place "modulo" computation structural equations and concurrently.

We find the conceptual analogy between K and CHAM so

insightful, that we take the liberty to replace the equality symbol "=" in our computation structural equations with the symbol "$\rightleftharpoons$", which is used in CHAM for heating/cooling.